Context-Free Grammars

Context-free grammars are a much more powerful computation framework than regular languages. In this section we define them and provide some examples.

Reading

Section 2.1

Practice problems (page 128): 2.1, 2.3, 2.4, 2.5, 2.6, 2.9, 2.14, 2.15, 2.16, 2.17, 2.19, 2.21, 2.22, 2.23, 2.27, 2.28

Context-Free Grammars

Context-free grammars are based on a series of **productions** or **substitution rules**. These productions can contain **terminals** as well as **variables** or **non-terminals**. There is also a start variable. For example here is a grammar that reads palindromes (say the alphabet consists of only the letters a and b):

```
S –> aa
S –> bb
S –> aSa
S –> bSb
```

So in this case we have two terminal symbols, a and b, and one non-terminal S, which is also the start variable. Each "production" has a non-terminal on the left and a sequence of terminals and non-terminals on the right. That production essentially tells us that one way to obtain something of "type" S is by having the elements on the right.

We establish the a string is in the language if we can "produce" it via a series of such productions, starting from the start variable S. For example, the strings aa and bb are in the language because of the first two productions. But the string abba is also in the language because of the production S –> aSa and the fact that the S in the right-hand side can be obtained via hte production S –> bb.

A sequence of such productions is called a **derivation**. We usually visualize it via a **parse tree**, where each node represents a production tagged by the left-hand side of the production, and the node's leaves are the elements in the right-hand side of that production.

Note that in the above example we could simplify the grammar if we have a epsilon on the right side, representing the act of replacing S with nothing at all. This will also capture the empty string, which should be considered a palindrome:

```
S –> epsilon
S –> aSa
S –> bSb
```

Before we see a formal definition of context-free grammars, let us look at one more example. Here is a specification for a context-free language for a limited set of arithmetic expressions. The "alphabet" in this case is all digits, parentheses, and the symbols for addition and multiplication.

```
S –> N | S + S | S * S | ( S )
N –> x|y
```

The terminals are exactly the elements in the alphabet. We have 3 "non-terminals", namely S, N and P. S represents an "expression", "N" represents a number (and we don't explicitly look for a number but instead the variable names x, y, maybe representing integers and floating point numbers respectively). We have also used a vertical line to separate productions from the same left-side. For instance the "production" `N –> x|y` stands for the two productions `N –> x` and `N –> y`.

Question: Is the "string" `x + x * x` derived from this grammar? Is that possible in more than one way? Draw the corresponding parse trees.

Let us look at a formal definition:

A **context-free grammar** is a 4-tuple (V, Σ, R, S) where:

1. V is a finite set, called the **variables**.
2. Σ is a finite set, disjoint from V, called the **terminals**.
3. R is a finite set of **rules**, each rule consisting of a variable and a string of variables and terminals, or more formally
$$R \subset V \times (V \cup \Sigma)^*$$
4. A **start variable** $S \in V$.

If $u, v, w \in (V \cup \Sigma)^*$ are strings of terminals and variables, and $A \to w$ is a rule (i.e. $(A, w) \in R$), then we say that uAv **yields** uwv, and we write $uAv \Rightarrow uwv$.

We say that u **derives** v, and write $u \Rightarrow^* v$, if there is a sequence of one or more strings $u_1, \ldots, u_k$ where
$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

The **language of the grammar** is
$$L = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Ambiguity, leftmost derivations

Let us look back at our example from arithmetic expressions, and for simplicity we will consider a smaller grammar:

```
S –> x
S –> S + S
S –> S * S
S –> ( S )
```

Now consider the expression x+x*x. There are numerous ways to derive it. We can classify their differences into two "groups": They might be giving rise to the same parse tree, or they might not. For instance:

S
S ∗ S
S + S ∗ S
x + S ∗ S
x + S ∗ x
x + x ∗ x

S
S + S
S + S ∗ S
x + S ∗ S
x + S ∗ x
x + x ∗ x

S
S + S
x + S
x + S ∗ S
x + x ∗ S
x + x ∗ x

These are 3 different derivations. But two of them give rise to the same parse tree. It would be useful to separate these two ideas.

> Given a parse tree, there is a derivation for it called the **leftmost derivation**. It is obtained by at every step replacing the left-most variable. There is only one leftmost derivation for any given parse tree.

Here are two leftmost derivations for the expression x+x∗x:

S
S + S
x + S
x + S ∗ S
x + x ∗ S
x + x ∗ x

and

S
S ∗ S
S + S ∗ S
x + S ∗ S
x + x ∗ S
x + x ∗ x

Draw the two parse trees, make sure to notice the differences. Semantically this is very imporant: One parse tree represents our normal notion of how we would compute such an expression, by performing the multiplication first, the other represents performing the addition first.

This is a problem: In the current grammar we have two completely different ways to understand what x + x ∗ x means. The computer would not know which to choose. This is a defect in the grammar.

A string w is said to be derived **ambiguously** in a CFG G, if it has two or more different leftmost derivations. A grammar is **ambiguous** if it generates some string ambiguously.

Question: In our example above there is also ambiguity in deriving the string $x + x + x$. Draw the corresponding parse trees. What property of numbers is this ambiguity related to?

We can "fix" this some times by the introduction of more symbols into the grammar. For instance in the above example we could do:

```
S –> S + T | T
T –> T * F | F
F –> ( S ) | x
```

The idea here is that we use distinct symbols to represent *terms* and *factors*. An expression is a sum of one more more terms, and a term is a product of one or more factors, and the order is forced. In this example the only way we can derive via a leftmost derivation that x + x * x is in the language is via:

```
S
S + T
T + T
F + T
x + T
x + T * F
x + F * F
x + x * F
x + x * x
```

CFG from a regular language

It is easy to show that a regular language is also context-free. There are two approaches:

- Look at the DFA for the language.
  - Make a variable/nonterminal R_i for each state i of the DFA.
  - If there is a transition $\delta(i, a) = j$ in the DFA add a production rule $R_i \to aR_j$.
  - Add a production rule $R_i \to \epsilon$ for each accept state in the DFA.
  - The start variable is the one corresponding to the start state of the DFA.
- Start from a regular expression for the language.
  - For each "regular expression piece" r we have a variable/nonterminal R_r.
  - The start variable corresponds to the overall regular expression as one piece.
  - If a regular expression r is a union of two others s and t, add an appropriate rule $R_r \to R_s|R_t$.
  - If a regular expression r is a concatenation of two others s and t, add an appropriate rule $R_r \to R_sR_t$.

- If a regular expression r is the Kleene star of a regular expression s, add the rules $R_r \to \epsilon | R_r R_s$.
- If a regular expression r is just a terminal x, add a rule $R_r \to x$.
- If a regular expression r is for the empty string ϵ, add a rule $R_r \to \epsilon$.
- If a regular expression r is for the empty language $\emptyset$, we do not any rule. The lack of a rule means that if we end up with this variable, we cannot get rid of it, and hence no strings will be produced.

Chomsky Normal form

It is occasionally convenient to bring all CFGs to a "normalized" form. This is called the **Chomsky Normal Form**.

A grammar is in a Chomsky Normal Form if every rule is of the form:

$$A \to BC$$

$$A \to a$$

$$S \to \epsilon$$

i.e. the right-hand side is either a single terminal symbol or two non-terminal symbols, and we also allow the rule that derives the empty string from S, if that is in the language. But no other variables can derive ϵ.

Any language that is recognized by a CFG can also be recognized by a CFG in CNF.

Let us discuss the proof of this fact. The idea is simple: Replace rules from the original language with simpler rules, while patching up the grammar to preserve the language. There are three steps:

1. Add new start variable.
2. Eliminate ϵ-rules $A \to \epsilon$.
3. Eliminate *unit rules* $A \to B$.
4. Finally we convert the remaining rules.

Let's take it one step at a time, first of all by adding a new start variable S_0 and a rule $S_0 \to S$. This ensures that the start variable is not used anywhere else (i.e. doesn't appear on the right-hand-side of any production).

Now we want to remove all ϵ-rules. The idea is simple: If there is a rule $A \to \epsilon$, then we find all rules where there is an A in the right-hand side, then add a rule that is just like that one but with the A removed (this might amount to more than one new rule if A appeared more than once in the same rule). Finally, we remove the $A \to \epsilon$ rule.

Next, we want to eliminate unit rules. If there is a rule $A \to B$, then we look for all rules of the form $B \to u$, and for each we add a rule $A \to u$, unless we had previously removed that rule. We can then remove the $A \to B$ rule.

Lastly, we consider other types of rules. For instance suppose we have a rule $A \to u_1 u_2 u_3 \dots u_k$, where $k \geq 3$ and each u_i is a variable or symbol. Then we can replace this rule with a sequence of rules: $A \to u_1 A_1$, $A_1 \to u_2 A_2$, $\dots$, $A_{k-2} \to u_{k-1} u_k$. Here $A_1, \dots A_{k-1}$ are newly introduced variables.

If we have a rule $A \to u_1 u_2$ where u_1 and u_2 are variables or terminals, and one of them is a terminal, then we replace that terminal with a rule like $A_1 \to u_2$, adjusting the original rule.

Let us follow this method in the example of a small arithmetic expression grammar, which we expand a little to allow for digits/numbers/signs:

```
E –> E+T | T
T –> T*F | F
F –> YN
N –> D | ND
D –> 0 | 1
Y –> + | – | epsilon
```

We start by adding a new first state:

```
S –> E
E –> E+T | T
T –> T*F | F
F –> YN
N –> D | ND
D –> 0 | 1
Y –> + | – | epsilon
```

We will next remove any ϵ rules, namely the rule `Y –> epsilon`. To remove it, we need to go into all the places where `Y` was being used, and replace its possible occurences with ϵ. We then remove the `Y –> epsilon` rule. This leads us to:

```
S –> E
E –> E+T | T
T –> T*F | F
F –> YN | N
N –> D | ND
D –> 0 | 1
Y –> + | –
```

Next we need to remove "unit rules". We have a number of such rules: `E–>T` and `T–>F`, `S–>E`, `F–>N`, `N–>D`. We start with `E–>T`. We look at all derivations `T–>u` and add ones of the form `E–>u`. And we remove the `E–>T` term.

```
S –> E
E –> E+T | T*F | F
T –> T*F | F
F –> YN | N
N –> D | ND
D –> 0 | 1
Y –> + | –
```

Next we do the same for the rule `T–>F`. We need for each rule `F–>u` to add a rule `T–>u`:

```
S -> E
E -> E+T | T*F | F
T -> T*F | YN | N
F -> YN | N
N -> D | ND
D -> 0 | 1
Y -> + | -
```

Now we have the rule `E->F` that was created along the way:

```
S -> E
E -> E+T | T*F | YN | N
T -> T*F | YN | N
F -> YN | N
N -> D | ND
D -> 0 | 1
Y -> + | -
```

Now let's look back at the rule `N->D`, which is perhaps where we should have started. We find all productions of `D`, and add rules to `N` for them:

```
S -> E
E -> E+T | T*F | YN | N
T -> T*F | YN | N
F -> YN | N
N -> ND | 0 | 1
D -> 0 | 1
Y -> + | -
```

Now we do the same for the rule `F->N`:

```
S -> E
E -> E+T | T*F | YN | N
T -> T*F | YN | N
F -> YN | ND | 0 | 1
N -> ND | 0 | 1
D -> 0 | 1
Y -> + | -
```

Next is the rule `T->N`, and after it `E->N`:

```
S -> E
E -> E+T | T*F | YN | ND | 0 | 1
T -> T*F | YN | ND | 0 | 1
F -> YN | ND | 0 | 1
N -> ND | 0 | 1
D -> 0 | 1
Y -> + | -
```

Lastly, we have the rule `S->E`. We need to look at rules `E->u` and add `S->u` rules:

```
S -> E+T | T*F | YN | ND | 0 | 1
E -> E+T | T*F | YN | ND | 0 | 1
T -> T*F | YN | ND | 0 | 1
F -> YN | ND | 0 | 1
N -> ND | 0 | 1
D -> 0 | 1
Y -> + | -
```

This finishes the first phase, leaving us on the right with either terminals or rules that have at least two elements.

Next we need to look at all rules that have more than two elements, and break them down. We have many such rules. We will try to avoid duplication by reusing "added variables" where appropriate. A computer would probably not do that on a first pass (but could do so on a second pass).

```
S   –> EA_1 | TA_2 | YN | ND | 0 | 1
E   –> EA_2 | TA_2 | YN | ND | 0 | 1
T   –> TA_2 | YN | ND | 0 | 1
F   –> YN | ND | 0 | 1
N   –> ND | 0 | 1
D   –> 0 | 1
Y   –> + | –
A_1 –> +T
A_2 –> *F
```

Now we have to work on the terms that have two elements on the right side, but some of those are terminals. We again will avoid duplicates:

```
S   –> EA_1 | TA_2 | YN | ND | 0 | 1
E   –> EA_2 | TA_2 | YN | ND | 0 | 1
T   –> TA_2 | YN | ND | 0 | 1
F   –> YN | ND | 0 | 1
N   –> ND | 0 | 1
D   –> 0 | 1
Y   –> + | –
A_1 –> PT
A_2 –> MF
P   –> +
M   –> *
```

And there you have it! A Chomsky Normal form grammar! Simpler terms, but a lot harder to reason about.

Exercise: Do the same to the following grammar, that expresses palindromes:

```
S –> aSa | bSb | a | b | epsilon
```

If you avoid duplication you should end up with 4 new variables in addition to the new start state.